RMPP: The Reliable Message Passing Protocol*

Rolf Riesen
Sandia National Laboratories
Albuquerque, NM 87185-1110
rolf@cs.sandia.gov

Arthur B. Maccabe
University of New Mexico
Albuquerque, NM 87131-1386
maccabe@cs.unm.edu

Abstract

Large-scale clusters built out of commercial components face similar scalability obstacles as the massively parallel processors (MPP) of the 1980's. This is especially true when they are used for scientific computing. Their networks are the descendants of the MPP networks, but the communication software in use has been designed for wide-area networks with client/server applications in mind.

We present a communication protocol which has been designed specifically for large-scale clusters with a scientific application workload. The protocol takes advantage of the low error rate and high performance of these networks. It is adapted to the peculiarities of these MPP-like networks and the communication characteristics of scientific applications.

This paper only presents the protocol itself and the ideas behind it. We refer the reader to other publications for more information about scalability, performance, and usage of the protocol presented here.

1. Introduction

The advent of large clusters, based on commercially available components, has made it possible to carry out scientific computations much cheaper than using a custom-built massively parallel processor (MPP). It is tempting to extend the paradigm and also use standard workstation and PC-class software. While this may work for smaller clusters, a cluster with thousands of nodes requires a more thought-out approach to be scalable. This is especially true for its network and its communication protocols.

In this paper we describe a scalable protocol which is currently employed in Cplant™ [4]. The largest Cplant™ currently in existence has a size of about 1800 nodes. There are several reasons we decided not to use a standard protocol such as TCP/IP for Cplant™.

The networks employed by Cplant™ and other high-performance clusters are descendents of the networks that were used in MPPs in the 1980's. They have different characteristics and are used differently than wide area networks or the world wide web. These networks are not completely error free. But errors occur very infrequently and are detected by the network hardware. There is no need for software checksums or error correction codes. The physical distances are short and the networks achieve very high speeds. Retransmitting a one megabyte message once in a great while is acceptable.

The switches in these MPP-like networks are very simple. Usually messages are source routed and the switch does not contain any logic to decide which outgoing port to use. There is hardware flow control and CRC checks, but no mechanisms to interact with the switches such as ICMP messages in Ethernet switches. Messages are wormhole routed which eliminates the need for large buffers inside the switches. It also means that messages are not dropped when congestion occurs.

The network interfaces (NIC) of these networks can DMA data directly to and from user space, are often programmable, and could issue an interrupt to the host every few microseconds. It is therefore important to bundle interrupts or avoid them whenever possible. Unfortunately, these NICs are attached to relatively slow I/O buses and often have not very much memory available for buffering. This makes flow control a necessity.

Scientific applications have different communication characteristics than client/server applications. Communication occurs in the form of, often long, messages and many times in all-to-all patterns. Ideally, these messages flow from user buffers directly into the network and into a user buffer on the remote node. Avoiding memory to memory copies is important to keep CPU usage for protocol processing low and bandwidth high.

For Cplant™ we needed a flexible, scalable, light-weight protocol which guarantees reliability, but still delivers high-

*This work was supported by Sandia, a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC004-94AL85000.

performance in the usual, error free, case. It should be possible to run the protocol on the NIC or on the host side. The protocol should be message-based and allow delivery of messages directly into user space.

The following section describes he properties of RMPP, the Reliable Message Passing Protocol. In Section 3 we present the specification of RMPP. Section 4 concludes the paper with improvement suggestions and ideas for future work.

This paper only presents the protocol itself, the ideas behind it, and why we made certain design decisions. Please refer to [7, 5] and [8] for quantitative data, comparisons to other protocols, and how higher level layers can make use of RMPP.

2. A New Protocol

RMPP is a transport layer protocol originally designed to packetize and reliably transmit Portals 3.0 messages [2, 3]. It is meant to operate over a high-speed networks with low latency and high bandwidth, assumes that network errors are rare, and demands little CPU power.

The low network error rate makes it possible to optimize RMPP for the general case when no errors occur. RMPP is willing to pay a high price in the rare cases when it has to recover from a network error. This keeps the protocol and its implementation simple, and keeps protocol overheads for acknowledgements and control packets low.

RMPP is message-based which means it concerns itself with the reliable delivery of whole messages, not individual packets. For example, only whole messages are acknowledged, not individual packets. RMPP also understands that the first packet of a message is important to upper layers. It may contain upper level protocol headers that need to be processed before the data contained in the message can be received.

RMPP does not concern itself with congestion avoidance or control. The two end points of an individual message are poor locations to make decisions about the global state of congestion in the network. Furthermore, MPP switches simply delay packets during periods of congestion; eventually they will be delivered. Traditional protocols use dropped packets as an indicator for congestion, but that would be a poor metric in an MPP network. Flow control, on the other hand, is a part of RMPP.

All aspects of RMPP were designed with scalability to thousands of nodes in mind. For example, flow control must not lower the achievable cross-section bandwidth, even as the cluster grows. Also, the addition of more nodes must not significantly increase the amount of state kept at each node, or the time it takes to establish and tear down all connections.

Lastly, RMPP was also designed to be simple to understand and implement. This helps in debugging and testing, and has had the side effect of making RMPP modular such that portions of it can be off-loaded onto the NIC [8, 5].

2.1. Design Goals

RMPP was designed for Cplant™ when we first realized that Myrinet [1] hardware was not quite as reliable as we had originally thought. RMPP is message based because it is supposed to provide a reliable transport layer for Portals 3.0. RMPP is not specific to Myrinet or Portals 3.0, but it does assume a high performance, MPP-like network with a very low error rate. RMPP should be easy to implement, have low CPU overhead and other low resource requirements, be compatible with OS and application bypass, and must scale to thousands of nodes. It can be implemented on the host side or inside the NIC.

2.2. Characteristics

Because RMPP is designed for networks with a very low error rate and because it transports messages, it only acknowledges whole messages not individual packets. If data can be DMAed directly into host memory, then only one interrupt per message is required. If RMPP runs on the NIC together with the decision portion of the upper layer, then interrupts can be avoided altogether. The decision where to put the data is made on the NIC, and RMPP transfers the data directly into user space.

RMPP retransmits all of the data of a message if any of it did not arrive intact. This is a huge overhead, but because errors are rare this cost is paid very infrequently. In the normal case, the small number of acknowledgement packets makes RMPP very efficient.

Since the switches should not drop packets when congestion occurs, and because it would be very difficult for RMPP to learn in a short enough time about the global state of the system and where the congestion is located, it does not make sense for RMPP to try to control congestion. On the other hand, RMPP does provide flow control.

Data has to be staged in NIC memory on its way to and from host memory and to and from the network. Since the two sides operate asynchronously and at different speeds, flow control is necessary. In our current implementation RMPP runs inside the kernel on the host side. The control program in the NIC is very simple and uses buffers in host memory. Therefore, it is RMPP which has to manage these buffers. This is especially important for incoming data. Many nodes sending large messages to the same node could easily exhaust the available buffers.

When RMPP acknowledges the first packet and grants the request to send more data, it also passes along how much data can be sent at this time. This number is based on the amount of other data currently flowing into that receiver. Note that this method is scalable, since the amount of data flowing into a given node is controlled locally by the receiver. There is no global state or a fixed number of allowances which can become scarce if more nodes are added to the network.

RMPP uses implicit connection establishment when it sends the first packet of each message. That packet contains the first portion of the user data as well as a request to send more data (if there is any). This approach creates one connection per message in transit. The resource requirements per node are not going to increase if the systems grows. This is not the case for systems in which each node creates a static connection to each other node in the system.

2.3. Implementation Notes

We mentioned above that RMPP keeps state only for the duration of a message. In addition, there is some state, corresponding to two integers per node in the system, that is persistent and grows linearly with the system.

In order to guarantee message ordering between any two nodes, RMPP must remember the last message number sent and received from every other node. RMPP will only accept a new message with the number it expects next. All others are rejected and will have to be retransmitted later by the sender. This algorithm is applied to the first packet of each message. This guarantees arrival ordering but still allows multiple messages in transit; including short messages while a longer message is still being transferred.

RMPP is not full-duplex like TCP. However, since connection setup is implicit, nodes can send each other messages simultaneously. The two data streams will not share a single connection, but data can flow in both directions nevertheless.

In order to guarantee reliability and keep the protocol as simple as possible, RMPP has a very clear division of tasks between the sender and the receiver of a message. The receive side prevents errors by dropping any packet that is bad, out of sequence, or disrupts message order. The send side is responsible for recovery by using timeouts. When the sender detects that a request for data transfer has not been granted in a given amount of time, or an acknowledgement is outstanding, it retransmits the request. In most cases this will result in the whole message being sent again.

This separation of tasks wastes some time and is not very efficient for recovery of lost data. However, errors are rare and the network is fast. The waste and inefficiency does not occur often and will not last long. In return, the protocol remains simple and efficient in the case when all goes well.

The simplicity of RMPP and its well-structured behavior makes it possible to move portions of the protocol onto the NIC. For example, once the connection has been established, future requests to send more data can be handled by the NIC. Observing flow control restrictions, the NIC can grant the sender to transmit more packets instead of interrupting the host to handle this control packet.

3. Specification

We presented justifications for the presence or absence of features in RMPP in earlier sections of this paper. In this section we provide a detailed description of RMPP.

Traditionally, a protocol specification includes information about the maximum packet length (MTU) and byte ordering. We omit that in the specification for RMPP. The reason is that RMPP is designed as an intra-cluster protocol. All nodes within the cluster speak the same version of the protocol and agree at boot time on these base parameters. For example, in our Myrinet-based Cplant™, the MTU for RMPP is just below 8192 bytes, the page size of an Alpha processor. RMPP over Ethernet on a x86 system uses a little less than 4096 bytes.

Security, another important aspect of wide area networks, is also not addressed by RMPP. RMPP trusts the packet header information, since it is provided by a trusted peer layer. The cluster as a whole system needs to be protected from the outside. The RMPP layer is not directly accessible from outside the cluster and has not direct route beyond the cluster network boundaries.

3.1. Header Format

Figure 1 shows the RMPP header format.

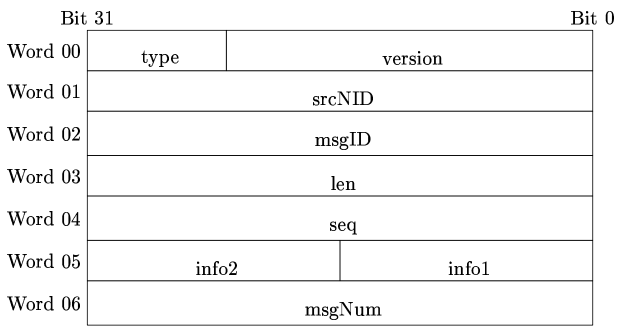

Figure 1. RMPP packet header

The `version` field is used to make sure both ends of a connection agree on the syntax and semantics of the protocol in use. Packets with non-matching version numbers are dropped.

The packet type is encoded in the `type` field. We explain the various types in the next section.

In order to respond to new message send requests, RMPP must know where the message is coming from. The `srcNID` field is used to identify the sender. RMPP assumes that each node in the system is assigned a unique number. Typically the nodes are numbered by the runtime system from $0\ldots(n-1)$, where n is the number of nodes in the system. A layer below RMPP is responsible for translating this number into a valid network address or a route to the correct destination.

Each message is assigned a unique, system wide, message identifier. Each packet that is part of this message, is marked with the same identifier as well as the same source node identifier, independent of which direction the packet is traveling. This identifier is stored in the `msgID` field and, together with the `srcNID` field, is used to locate state information on the receive as well as the send side.

The `len` field contains the length of the whole message in bytes. The sequence field (`seq`) is used to verify ordering of data packets in longer messages. The `info1` and `info2` fields contain packet type specific information and will be discussed further in the next section.

Each message between two specific nodes in a pair is numbered. RMPP uses that number to preserve message ordering. It is stored in the `msgNum` field of the header. Section 3.8 gives more details about message ordering.

3.2. Packet Types

The current implementation of RMPP uses ten different packet types. They are listed in Table 1.

Table 1. RMPP packet types

Type	Description
RTS <data>	Request to send for long msg
LAST_RTS <data>	Request to send for short msg
CTS [n]	Clear to send n data pkt
DATA [s] <data>	Data pkt with sequence number
STOP_DATA [s] <data>	Data pkt; last of granted block
LAST_DATA [s] <data>	Data pkt; last of msg
MSGEND	Msg successfully received
MSGDROP	No more data wanted; finish
GCH	Garbage collection hint
NULL	Terminate transfer

An `RTS` packet is used to start a message which is longer than one packet size. For messages which fit entirely into the first packet, a `LAST_RTS` packet is used to send the message. Both packet types contain user data, including upper level protocol headers. In our current implementation, the RMPP packet header is followed by a 64-byte Portals 3.0 header, and then user data. RMPP does not have any knowledge about the Portals 3.0 header. It simply passes the first packet to the layer above for processing.

A `CTS` packet is used to acknowledge the receipt of a `RTS` packet and to grant the sender the right to transmit up to n data packets. The allowance n is stored in the `info1` field of the header. The value of n is determined by the receiver and used to control flow into the receiving node. It is always $n \geq 1$. If few messages are flowing into the receiving node simultaneously, n is set to a high value. If many nodes are sending to this receiver, then n is lowered to throttle incoming traffic.

Once a `CTS` packet has been received by the sender, it will transmit up to n data packets. These packets are of type `DATA` and contain a sequence number s which is stored in the `seq` field of the header. The `info1` field contains n to allow the receiver to update its outstanding packet count. The `info2` filed is used to enumerate the number of times the sender has tried to send this message. Usually it is zero.

The last data packet of a block is either `STOP_DATA`, if the sender has more data but ran out of allowed packets to send, or `LAST_DATA` if this concludes the message.

When a receiver has received all data of a message, it sends a `MSGEND` packet to the sender. If, after an `RTS` packet, the upper layers inform RMPP that the message is not wanted, then RMPP sends a `MSGDROP` packet to signal the sender to complete this message without any further data transmission.

A `MSGEND` or `MSGDROP` packet signals the successful receipt of all the data the receiver wanted to accept. The sender can now release all buffers and state entries that were needed for this message. The sender transmits a `GCH` packet to the receiver, so the receiver can release its state information for this message as well. The `GCH` packet is only a hint to do the clean-up early. The receiver will reuse resources allocated to a message after it has sent a `MSGEND` or `MSGDROP` and a sufficiently long time has passed. A few lost `GCH` packets do not hamper RMPP. However, `GCH` is necessary to reclaim unused state space on the receiver side. Without `GCH` state space consumption might exceed reasonable limits on the receive side.

If a `MSGEND` or `MSGDROP` is lost, the sender will eventually resend the first data packet. The receiver will recognize that the all of the data has already been received and simple resend the lost `MSGEND` or `MSGDROP`.

The `NULL` packet is used to abort a current message. It is sent in place of a `RTS` or `RTS_LAST` retry when a sending process gets killed or wishes to abort the current transfer. This is necessary to keep the message numbering between two nodes synchronized and avoid deadlock [6, p. 92].

3.3. Execution Flow

RMPP treats each message separately. It maintains state for each message as long as it is being transferred. The following state diagrams therefore apply to each individual message. Several messages can be transferred simultaneously. The state information kept on the two communicating nodes must include information about a message's progress through the following state machines.

Figure 2 shows the send side state diagram for short messages. Longer messages are separated out for clarity in the state diagram in Figure 3. Figure 4 shows the state diagram for the receive side. It handles short and long receives.

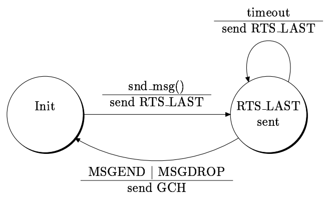

Figure 2. RMPP state diagram for short sends

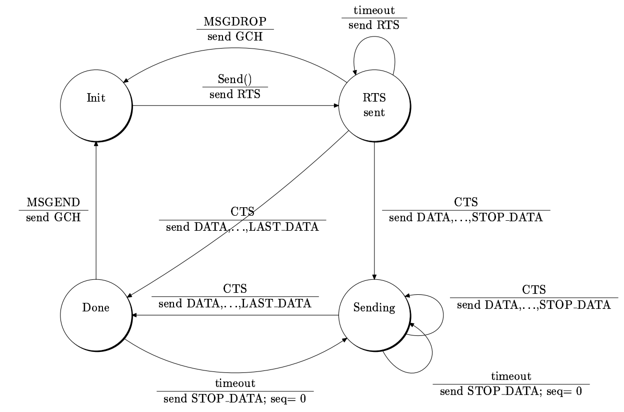
MSGDROP / send GCH
timeout / send RTS
Init
Send() / send RTS
RTS sent
MSGEND / send GCH
CTS / send DATA,...,LAST_DATA
CTS / send DATA,...,STOP_DATA
CTS / send DATA,...,LAST_DATA
Done
Sending
CTS / send DATA,...,STOP_DATA
timeout / send STOP_DATA; seq= 0
timeout / send STOP_DATA; seq= 0

Figure 3. RMPP state diagram for long sends

In all three state diagrams unexpected packets are simply dropped. Figure 4 does not show the transition from the `done` state back to the `init` state when a `GCH` packet has been lost. In our implementation that transition occurs ten minutes after a `MSGEND` or `MSGDROP` has been sent but no `GCH` has been received. This timeout is arbitrary. It needs to be large enough to ensure that any delayed `GCH` packet will come in before the timeout, and small enough to avoid excessive storage use for state data.

Figure 5 gives an example of a short message transfer. The sending node sends an `RTS_LAST` packet. That packet includes all the user data for this message. The receiver accepts the data and acknowledges the message with a `MSGEND`. The sender can now release any state information it had for this message. It sends a `GCH` packet to let the receiver know it can discard its state information for this message as well.

Figure 6 shows an example of a longer message transfer.

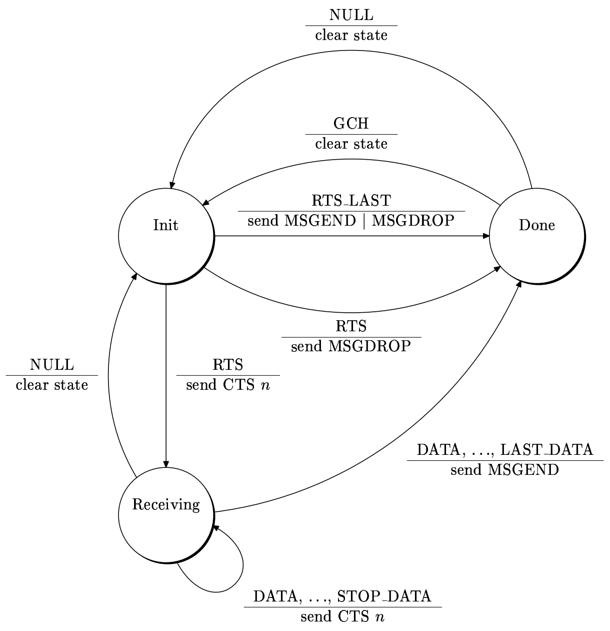

Figure 4. RMPP state diagram for receives

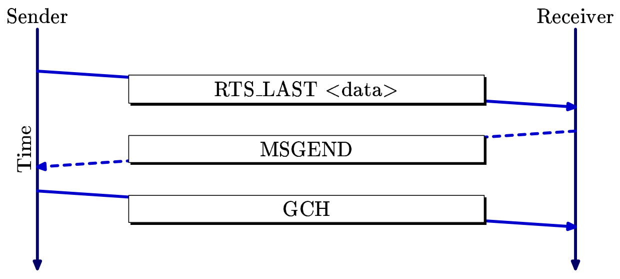

Figure 5. RMPP timing diagram for short messages

The sending node sends an `RTS` packet including some user data. The receiver acknowledges the `RTS` with a `CTS` packet and allows the sender to transmit n_1 data packets. Usually, the number of packets a receiver grants to a sender remains the same for each `CTS` packet. In our current implementation that parameter is set to 16. However, if a receiver suddenly gets bombarded with message requests, subsequent `CTS` packets might have different values for n.

When the sender receives the first `CTS` packet, it sends n_1 data packets. The numbers in square brackets indicate the sequence number used. The last data packet in this block is of type `STOP_DATA`. This tells the receiver that the number of allowed data packets has been exhausted and another `CTS` is necessary to continue the transfer.

The receiver sends the second `CTS` and allows the sender to transmit n_2 data packets. The sequence numbers for this block of data packets starts at $n_1 + 1$ and ends at $n_1 + n_2$.

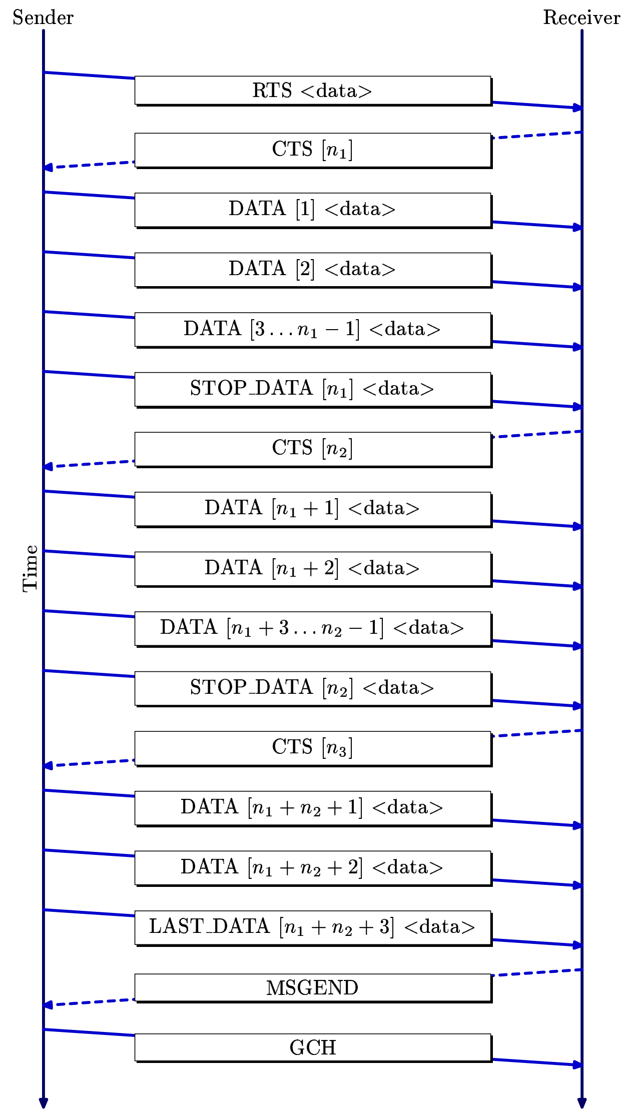

Figure 6. RMPP timing diagram for long messages

Again, the last packet in the block is of type STOP_DATA to indicate to the receiver that another CTS is necessary to proceed.

The receiver sends the third and final CTS packet, granting the sender n_3 data packets. The sender has only enough data left to fill three data packets. They are assigned sequence number $n_1 + n_2 + 1$ through $n_1 + n_2 + 3$. The last data packet is of type LAST_DATA indicating to the receiver that the sender has sent all data. The receiver acknowledges the receipt of all data using a MSGEND packet. The sender sends a GCH packet to inform the receiver that it can now clean-up any remaining state information for this particular message.

3.4. Connection Management

RMPP connections do not have to be established before data can be sent. RMPP creates a connection for each message implicitly with the first packet (the RTS or RTS_LAST) that is sent. The connection is confirmed by the receiver with a CTS or a MSGEND (or a MSGDROP) for short messages. The final MSGEND or MSGDROP closes the implicit connection.

Therefore, connections are very short lived. While a connection is active, both sides need to maintain some state, but each node only keeps state about currently ongoing messages. After both sides agree that a message has been successfully transmitted, all state associated with that message disappears.

In our implementation, each message is assigned a system-wide unique message identifier. Every packet that is sent on behalf of that message carries that identifier in the msgID field of the packet header. The message identifier is generated by the sending node just before a RTS or RTS_LAST packet is sent and is a local count which gets incremented for each message. Together with the srcNID field, this uniquely identifies each packet in the system.

We use the message and the source node identifier to locate state information which is stored in lists. We maintain three lists. State information for a message goes into the send_pending queue when the first RTS or RTS_LAST packet has been sent. Upon receipt of a CTS packet for that message, the queue entry gets moved to the sending queue (or removed if a MSGEND or MSGDROP is received.) On the receive side a new entry in the receiving queue gets created when a RTS or RTS_LAST arrives.

3.5. Reliability

RMPP's reliability is based on messages. Only whole messages are acknowledged, not individual packets. RMPP assumes that errors are rare and sending unnecessary acknowledgments for individual packets is more wasteful than re-sending a whole message once in a great while.

The sending node of a message is the active partner in error recovery. It sets up a timer and monitors progress of the packet exchange. If a receiver's reply has not been received when the timer expires, the appropriate packet is re-sent by the sender. If no MSGEND or MSGDROP has been received after sending an RTS_LAST or RTS when the timer expires, the RTS (or RTS_LAST) is re-sent. Once the sender has received the first CTS and the timer expires because no further CTS, MSGEND, or MSGDROP packets have been received, the sequence number is reset to 1 and one packet full of data is sent again.

The receiver is passive through error recovery. When it notices that the sender has restarted a message, because the sequence number has been reset back to 1, it synchronizes its own sequence number and starts receiving and acknowledging data packets again. The receiver does this by sending a CTS and allowing the sender to transmit multiple data packets again. In all other cases when it receives a packet that is out of sequence it simply drops it. The sender will eventually time out and restart the message.

This logic has a very high cost to recover lost or corrupted packets. However, it keeps the code, especially on the receive side, very simple. If recoveries occur infrequently, the lost time is not a factor.

In order to accomplish data integrity, RMPP relies on the layer below it to discard packets that have been corrupted. In our implementation the Myrinet NICs have built-in CRC checking. The control program running on the NIC flags packets which have a bad CRC. These packets will not be delivered to the RMPP layer.

Periodically, RMPP checks if there are pending sends which have not progressed in a long while. For each destination for which it finds an RTS or RTS_LAST it resends the oldest message; i.e. the message with the lowest message number.

3.6. Interface to Upper Layers

When RMPP receives an RTS or RTS_LAST packet it passes a pointer to the packet to the layer above. That layer is expected to determine where the data contained in the message needs to go. In our implementation, the layer above is the Portals 3.0 module. It could be any message-based delivery mechanism, though. Portals 3.0 considers the first 64 bytes of the first packet to be a message header. Using that, and information submitted to the Portals 3.0 module earlier by the user, it makes a decision where in user space the data needs to be delivered, and how much of the total message it is willing to accept.

Note that RMPP itself has no knowledge of the upper-layer headers. As long as these headers fit into the first packet, RMPP needs to concern itself only with its own small packet header.

The upper layer then calls back into the RMPP module with destination buffer information such as location and length. At that time the RMPP module generates a CTS and lets the sender know how much data of that message should be sent. The data which was transmitted with the first RTS packet is then copied into the user buffer.

In the case of a RTS_LAST the call-back into the RMPP module causes the data to be copied (or DMAed in a better implementation) into the user buffer. As soon as that is done, a MSGEND is sent back to the sender.

Figure 7 shows how RMPP ties into the packet module and the Portals 3.0 module. In our implementation the packet module is responsible for interacting with the NIC to send and receive individual packets. If Portals 3.0 wants to send a message, it calls the RMPP send function (1). This causes the RMPP protocol to call into the packet module (2) to send (and receive) a bunch of packets. When RMPP receives a MSGEND or MSGDROP packet, it calls lib_finalize() (A).

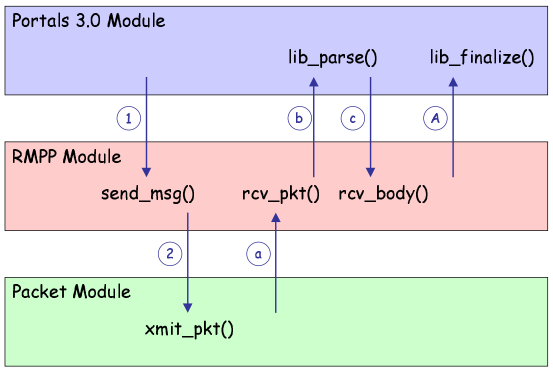

Figure 7. RMPP call-back mechanism

When a packet arrives at the packet module, it calls RMPP's rcv_pkt() function (a). If this is the first packet of a new message, RMPP makes an up-call (b) into the Portals 3.0 module. The Portals 3.0 module parses the header of the message and determines where the data needs to be delivered. It calls back into the RMPP module (c) to let it know where the data has to be delivered. Once all the data has been received, RMPP calls into the Portals 3.0 module one last time (A) to let it know that all the data has arrived.

Although we are talking about RMPP's relationship with the Portals 3.0 module, RMPP is independent of Portals 3.0. The upper layer simply needs to provide a function for RMPP to call when the first packet arrives. The upper layer must then decide where that data is supposed to go and call RMPP's rcv_body() function to give it that information.

Notice that RMPP itself does not perform any buffering. In our implementation, the packet module below RMPP uses buffers to stage incoming and outgoing packets. RMPP uses its own flow control mechanism to help manage those buffers. User data resides in space managed by the layers above RMPP. In our implementation RMPP copies data to and from these user buffers into or from the buffers in the packet module. These copies could be avoided by simply passing pointers, if the packet module and the control program running on the NIC were able to deal with data residing in user space.

3.7. Flow Control

We have mentioned flow control several times already. The main mechanism RMPP uses to implement flow control is to use packet allowances in its CTS packets. Before a CTS is sent, the receiving node evaluates how many packets it has currently granted to other incoming streams. It will grant at least one packet to assure progress for every message. As the granted data packets come in, the pool of available allowances increases again and future send requests may receive higher allotments in their CTS.

The code to determine the allowance sent out in a CTS packet is shown in Figure 8. In the current implementation at least one packet is granted, but never more than 16 (MAX_DATA_PKTS). The total pool of allowances (MAX_RCV_BUF) is set to 2048 in our implementation, since we have 2048 receive buffers available. NODES is currently set to 2048, reserving that many receive buffers for new connections from other nodes. The variable outstanding keeps track of the total number of data packets granted but not received yet. These values seem to be appropriate for our hardware in a 1800 node system, but need to be adjusted for other systems. If RMPP were running on the Myrinet card itself, these buffers could be eliminated, since Myrinet has hardware flow control and the data can be delivered directly into user space buffers.

```
n = MAX_RCV_BUF - NODES - outstanding;
if (n > MAX_DATA_PKTS) {
    n = MAX_DATA_PKTS;
} else if (n < 1) {
    n = 1;
}
outstanding += n;
```

Figure 8. Packets granted in a CTS

This strategy limits the incoming flow of data into a node when necessary and prevents buffer overflow. The decision how much data can flow into a node is made locally by the receiving node and does not affect other flows in the system. There is no global pool of allowances because that would limit scalability.

Another flow control problem occurs on the sending side. If a sending node transmits a lot of request to send packets to many different nodes, it might happen that all the receivers grant the sender the full number of allowances. This might cause the sender to overrun its own send buffers. For this reason RMPP has an upper limit on how many allowances it will grant in any one CTS messages. The receiver might have the capacity to receive all that data, but the sending node might not be able to handle it.

Even so, it is possible for a sending node to overflow its send buffers. RMPP tries to submit new RTS packets for transmission as soon as the upper layers request it. If the transmission layer below RMPP runs out of send buffers, further send requests by RMPP will fail. In that case RMPP simply enqueues the RTS (or RTS_LAST). When the timeout timer expires it will attempt to send it again.

3.8. Message Ordering

RMPP guarantees message ordering to the layers above. It uses the msgNum field in the packet header to do that. It is also necessary to keep track of the last message number sent and received about every node RMPP is communicating with. This is the only state information that persists beyond the lifetime of a single message. Our current implementation uses two integers in an array for every node in the system. So, even for an 8000-node system two arrays of 32 kB each are sufficient.

Message ordering can only be guaranteed for messages transmitted between two given nodes. (Message arrival from multiple nodes is non-deterministic.) Message ordering is based on message arrival, not message completion. As soon as the RTS of a new message arrives, RMPP informs the upper layers of that arrival (and receives placement information for the data of that message). A long message may not have completed yet when a RTS_LAST for a shorter message arrives. That shorter message may complete before all the packets of the larger message have been transmitted. However, arrival ordering remains intact because the arrival of the first packet of each message, independent of size, is communicated to the upper level in arrival order.

4. Improvements and Future Work

RMPP evolved out of the need to make message passing on Cplant™ more reliable. After studying the protocol we have several ideas to improve the protocol and its implementation.

The MSGDROP packet type is not really necessary. If a receiver does not want the message, it can send a MSGEND and abort the message early this way. Similarly, the RTS and RTS_LAST packet types can be combined, since the message length is part of the header and the number of bytes in the first packet is known as well (either by the lower layer passing it along, or the two sides knowing the MTU size in use.) The msgNUM and msgID fields could be combined.

Another possible improvement is to let the receiver send NAK packets when it detects errors. For example, a receiver might notice a dropped data packet because of a jump in the sequence number. Instead of waiting for the sender to time out, the receiver could send a NAK packet prompting the sender to start the retransmission earlier. A selective NAK would be a variation on this where the receiver requests the retransmission of specific data packets.

Doing that would complicate the protocol and its implementation. Our studies indicate that overall performance would not improve considerably in the network environment for which RMPP was designed. However, more studies are necessary to evaluate all aspects of these trade-offs.

5. Conclusions

RMPP is a simple communication protocol which is scalable to thousands of nodes. It is specifically suitable for low-error-rate, high-speed networks. This paper only presents the protocol and the ideas behind it. Performance and scalability measurements are provided in [7, 5] and [8]. The protocol is very modular and portions of it can be offloaded onto a programmable NIC. It is portable and is currently running over Myrinet and Ethernet.

Work is under way to characterize the modularity in more detail and explore what portions of a protocol and the message delivery layer above it should run on the host or the NIC. We have also identified a few improvements which will make RMPP even more simple. This helps with new implementations, testing, and verification.

Acknowledgements

Several people have helped us by reading drafts of this document and through insightful discussions. We would like to thank Jeff Ogden, Mike Levenhagen, Ron Brightwell, Kevin Pedretti, Wenbin Zhu.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local-area-network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[2] R. Brightwell, T. Hudson, R. Riesen, and A. B. Maccabe. The Portals 3.0 message passing interface. Technical report SAND99-2959, Sandia National Laboratories, 1999.

[3] R. Brightwell, R. Riesen, B. Lawry, and A. B. Maccabe. Portals 3.0: Protocol building blocks for low overhead communication. In *Workshop on Communication Architecture for Clusters CAC'02*, pages 0–0, Fort Lauderdale, Florida, Apr. 2002.

[4] The Cplant Project Homepage, 2001. `http://www.cs.sandia.gov/cplant/`.

[5] A. B. Maccabe, W. Zhu, J. Otto, and R. Riesen. Experience in offloading protocol processing to a programmable nic. In *IEEE Cluster 2002*, 2002.

[6] S. D. Osterman. *Reliable Message Transport for Network Communication*. PhD thesis, Purdue University, May 1994.

[7] R. Riesen. *Message-Based, Error-Correcting Protocols for Scalable High-Performance Networks*. PhD thesis, University of New Mexico, Albuquerque, New Mexico, July 2002.

[8] W. Zhu. OS-bypass investigation and experimenting. Master's thesis, University of New Mexico, Albuquerque, New Mexico, Mar. 2002.